Internet Congestion Control Research Group
Internet-Draft
Intended status: Experimental
Expires: September 22, 2016

M. Welzl
S. Islam
K. Hiorth
University of Oslo
J. You
Huawei

March 21, 2016

TCP in UDP
draft-welzl-irtf-iccrg-tcp-in-udp-00

Abstract

This document specifies a method to encapsulate multiple TCP connections using only one UDP port number pair. Doing so allows for a relatively easy implementation of coupled congestion control for the TCP connections. This can have several performance benefits, and it makes it possible to precisely assign a share of the congestion window to the connections based on priorities. It also enables use of UDP-based NAT traversal techniques, and it can act as a framework for experimentation with novel changes to the TCP standard.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Copyright Notice

Table of Contents

1. Introduction

Note that this document is written in a style that should facilitate quick reading by focusing on the key changes from prior similar proposals. A future version of this document will provide more details about the parts that are "inherited" from such prior work.

TCP-in-UDP (TiU) is based on [Che13]. It differs from it in that:

o Other than [Che13], TiU encapsulates multiple TCP connections using the same UDP port number pair. TCP port numbers are preserved; a single well-known UDP port is used for TiU. If TiU is implemented in the kernel, this allows using normal TCP sockets, where enabling the usage of TiU could be done via a socket option, for example.

o The header format is slightly different to allow representing a TCP connection with a few bits that are encoded across the original TCP header’s "Reserved" field and the URG (Urgent) flag to encode a Connection ID. With this encoding, similar to the encapsulation in [Che13], the total TiU header size does not exceed the original TCP header size.

o A (TiU-encapsulated) TCP SYN uses a newly defined TCP option to establish the mapping between a Connection ID and the original TCP port number pair.

o A method to couple the congestion controls of the TCP connections is presented. This coupling can have various performance benefits (explained in detail in Section 6) and allows to precisely allocate a desired share to one of the coupled TCP connections based on a priority from the application. Coupled congestion control is possible in TiU because the common preceding UDP header makes it reasonable to assume that the connections traverse the same network bottleneck. This is not necessarily a correct assumption when the outer header’s port numbers differ due to mechanisms like Equal-Cost Multi-Path (ECMP). Note that ECMP can have performance benefits which TiU eliminates. This trade-off is also discussed in Section Section 6.

o This document provides some new and/or somewhat different explanations: Section 4 discusses how TiU support can work with preceding extra information such as a SPUD header ([I-D.hildebrand-spud-prototype]) without exceeding the MTU and elaborates on a possible method of implementating TiU including robust "Happy Eyeballing".

TiU inherits all the benefits of [Che13] and a preceding similar proposal, [Den08]. It adds potential benefits that are due to coupled congestion control, and it adds the potential disadvantage of not being able to benefit from ECMP. In short, the benefits and features of TiU that are already explained in detail in [Che13] and [Den08] are:

o To establish direct communication between two devices that are both behind NAT gateways, Interactive Connectivity Establishment (ICE) [RFC5245] is used to create the necessary mappings in both NAT gateways, and ICE can have higher success rates using UDP [RFC5128].

o TCP options, as required for Multipath TCP [RFC6824], for example, are expected to work more reliably because middleboxes will be less able to interfere with them.

- o Because the packet format allows the first octet to be in the range 0x0-0x3 (as is the case for a STUN [RFC5389] packet, where the most significant two bits are always zero), the UDP port number pair used by TiU can be used to exchange STUN packets with a STUN server that is unaware of TiU.

- o Following the method described in [Che13] and [Den08], other transport protocols than TCP (e.g., SCTP) could be UDP-encapsulated in a similar fashion. With TiU, the same outer UDP port number pair could be used for different encapsulated protocols at the same time.

[Che13] also lists a disadvantage of UDP-encapsulating TCP packets: because NAT gateways typically use shorter timeouts for UDP port mappings than they do for TCP port mappings, long-lived UDP-encapsulated TCP connections will need to send more frequent keepalive packets than native TCP connections. TiU inherits this problem too, although using a single five-tuple for multiple TCP connections alleviates it by reducing the chance of experiencing long periods of silence.

2. More related work

The TCPMUX mechanism in [RFC1078] multiplexes TCP connections under the same outer transport port number; it does however not preserve the port numbers of the original TCP connections, and no method to couple congestion controls is described in [RFC1078].

TiU’s congestion control coupling follows the style of RTP application congestion control coupling in [I-D.ietf-rmcat-coupled-cc] which is designed to be easy to implement, and to minimize the number of changes that need to be made to the underlying congestion control mechanisms. This method was shown to yield several benefits in [fse]. TiU’s congestion control requires slightly deeper changes to the TCP’s congestion control, making it harder to implement than [I-D.ietf-rmcat-coupled-cc], but it is still a much smaller code change than the Congestion Manager [RFC3124].

Combining congestion controls as TiU does it has some similarities with Ensemble Sharing in [RFC2140], which however only concerns initial values of variables used by new connections and does not share the congestion window (cwnd), which is the variable of interest in TiU. The cwnd variable is shared across ongoing connections in [ETCP] and [EFCM], and the mechanism described in Section 5 resembles the mechanisms in these works, but neither [ETCP] nor [EFCM] address the problem of ECMP.

Coupled congestion control has also been specified for Multipath TCP [RFC6356]. MPTCP's coupled congestion control combines the congestion controls of subflows that may traverse different paths, whereas TiU builds on the assumption that all its encapsulated TCP connections traverse the same path. This makes the two methods for coupled congestion control very different, even though they both aim at emulating the behavior of a single TCP connection in the case where all flows traverse the same network bottleneck.

3. Specification

TiU uses a header that is very similar to the header format in [Den08] and [Che13], where it is explained in greater detail. It consists of a UDP header that is followed by a slightly altered TCP header. The UDP source and destination ports are semantically different from [Den08] and [Che13]: TiU uses a single well-known UDP port, and multiple TCP connections use the same UDP port number pair. The encapsulated TCP header is changed to fit into a UDP packet without increasing the MSS; this is achieved by removing the TCP source and destination ports, the Urgent Pointer and the (now unnecessary) TCP checksum. Moreover, the order of fields is changed to move the Data Offset field to the beginning of the UDP payload. This allows using it to identify other encapsulated content such as a STUN packet: for TCP, the Data Offset must be at least 5, i.e. the most-significant four bits of the first octet of the UDP payload are in the range 0x5-0xF, whereas this is not the case for other protocols (e.g., STUN requires these bits to be 0). The altered TCP header for TiU is shown below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data | Conn   |C|E|C|A|P|R|S|F|                               |
| Offset|  ID    |W|C|I|C|S|S|Y|I|            Window             |
|       |        |R|E|D|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      (Optional) Options                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Encapsulated TCP-in-UDP Header Format (the first 8 bytes are the UDP header)

Different from [Den08] and [Che13], the least-significant four bits of the first octet and a bit that replaces the URG bit in the next octet together form a five-bit "Connection ID" (Conn ID). TiU maintains the port numbers of the TCP connections that it encapsulates; the Connection ID is a way to encode the port number information with a few unused header bits. It uniquely identifies a port number pair of a TCP connection that is encapsulated with TiU. Using these five bits, TiU can combine up to 32 TCP connections with one UDP port number pair.

The TiU-TCP SYN and SYN/ACK packets look slightly little different, because they need to establish the mapping between the Connection ID and the port numbers that are used by TiU-encapsulated TCP connections:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |Re-    |C|E| |A|P|R|S|F|                               |
| Offset|served |W|C|0|C|S|S|Y|I|            Window             |
|       |       |R|E| |K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Encapsulated Source Port    | Encapsulated Destination Port |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Options                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: Encapsulated TCP-in-UDP SYN and SYN/ACK Packet Header Format

The Encapsulated Source Port and Encapsulated Destination Port are the port numbers of the TCP connection. To create this header, an implementation can simply swap the position of the original TCP header's port number fields with the position of the Data Offset / Reserved / Flags / Window fields.

Every TiU SYN or TiU SYN-ACK packet also carries at least the TiU-Setup TCP option. This option contains a Connection ID number. On a SYN packet, it is the Connection ID that the sender intends to use in future packets to represent the Encapsulated Source Port and Encapsulated Destination Port. On a SYN/ACK packet, it confirms that such usage is accepted by the recipient of the SYN. A special value of 255 is used to signify an error, upon which TiU will no longer be used (i.e., the next packet is expected to be a non-encapsulated TCP packet). The TiU-Setup TCP option is defined as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Kind          |    Length     |     ExID                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Connection ID |
+-+-+-+-+-+-+-+-+
```

Figure 3: TiU Setup TCP Option

The option follows the format for Experimental TCP Options defined in [RFC6994]. It has Kind=253, Length=5, an ExID that is with value TBD (see Section 8) and the Connection ID. The Connection ID is an 8-bit field for easier parsing, but only values 0-31 are valid Connection IDs (because the Connection ID in non - SYN or SYN/ACK TiU packets is only 5 bit long).

4. Protocol operation and implementation notes

There can be several ways to implement TCP-in-UDP. The following gives an overview of how a TiU implementation can operate. This description matches the implementation described in Section 7.

A goal of TiU is to achieve congestion control coupling with a simple implementation that minimizes changes to existing code. It is thus recommendable to implement TiU in the kernel, as a change to the existing kernel TCP code. The changes fall in two basic categories:

- Encapsulation and decapsulation: this is code that should, in the simplest case, operate just before a TCP segment is transmitted. Based on e.g. a socket option that enables/disables TiU, the TCP segment is changed into the TiU header format (Figure 1). In case it is a TCP SYN or TCP SYN/ACK packet, the header format is defined as in Figure 2, and the TiU-Setup TCP option is appended. This packet is then transmitted. For decapsulation, the reverse mechanism applies, upon reception of a UDP packet that uses destination port XXX (TBD, see Section 8). Both hosts keep a list of encapsulated TCP port numbers and their corresponding Connection IDs. In case a SYN packet requests using a Connection ID that is already reserved, an error (Connection ID value 255 in the TiU Setup TCP option) must be signified to the other end in a TiU-encapsulated TCP SYN/ACK, and encapsulation must be disabled on all further TCP packets. Similarly, when receiving a TiU SYN/ACK with an error, a TCP sender must stop encapsulating TCP packets.

- Coupled congestion control: this is code that influences the congestion control of TCP. Section 5 describes a simple coupled congestion control algorithm that can be applied to couple TCP connections and assign them a share of the total congestion window that is based on a priority.

The TCP port number space usage on the host is left unchanged: the original code can reserve TCP ports as it always did. Except for the TiU encapsulation compressing the port numbers into a Connection ID field, TCP ports should be used similar to normal TCP operation. A TCP port that is in use by a TiU-encapsulated TCP connection must therefore not be made available to non-encapsulated TCP connections, and vice versa.

For each TCP connection, two variables must be configured: 1) TiU-ENABLE, which is a boolean, deciding whether to use TiU or not, and 2) Priority, which is a value, e.g. from 1 to 10, that is used by the coupled congestion control algorithm to assign an appropriate share of the total cwnd to the connection. Priority values are local and their range does not matter for this algorithm: the algorithm works with a flow’s priority portion of the sum of all priority values. The configuration of the two per-connection variables can be implemented in various ways, e.g. through an API option.

With these code changes in place, TiU can operate as follows, assuming no previous TiU connections have been made between a specific host pair and a client tries to connect to a server:

- An application uses an API option to request TiU operation. The kernel then sends out a TiU TCP SYN that contains a TiU-Setup TCP option. This packet header contains the encapsulated TCP port numbers (source port A and destination port B) and the Connection ID X.

- The server listens on UDP port XXX (TBD, see Section 8). Upon receiving a packet on this port, it knows that it is a TiU packet and decodes it, handing the resulting TCP packet over to "normal" TCP processing. The TiU-Setup TCP option allows the server to associate future TiU packets containing Connection ID X with ports A and B. The server sends its response as a TiU SYN-ACK.

- TCP operates as normal from here on, but packets are TiU-encapsulated before sending them out and decapsulated upon reception, using Connection ID X. Both hosts associate TiU packets carrying Connection ID X with a local identifier that matches ports A and B, just like they would associate non-encapsulated TCP packets with the same local identifier when seeing ports A and B in the TCP header.

- If an application on either side of the TiU connection wants to connect to a destination host on the other side and requests TiU operation, the kernel sends out another TiU TCP SYN, this time containing a different TCP source port number and either the same or a different destination port number (C and D), and a TiU-Setup TCP option with Connection ID Y. From now on, packets carrying Connection ID Y will be associated with ports C and D on both hosts. Otherwise, TiU operation continues as described above.

- o Now, because there are two or more connections available between the same host pair, coupled congestion control begins to operate for all outgoing TiU packets (see Section 5 for details). This is a local operation, applying the priority values that were configured to use for the TiU-encapsulated TCP connections.

Unless it is known that UDP packets with destination port number XXX (TBD, see Section 8) can be used without problems on the path between two communicating hosts, it is advisable for TiU implementations to contain methods to fall back to non-encapsulated ("raw") TCP communication. Such fall-back must be supported for the case of Connection ID collisions anyway. Middleboxes have been known to track TCP connections [Honda11], and falling back to communication with raw TCP packets without ever using a raw TCP SYN - SYN/ACK handshake may lead to problems with such devices. The following method is recommended to efficiently fall back to raw TCP communication:

- o After sending out a TiU SYN packet, additionally send a raw TCP SYN packet.

- o After sending out a TiU SYN/ACK packet, additionally send a raw TCP SYN/ACK packet.

- o Upon receiving a TiU SYN packet, after responding with a TiU SYN/ACK packet and raw TCP SYN/ACK packet, immediately store the encapsulated port numbers and Connection ID. As long as a TiU connection is ongoing, ignore any additional incoming TCP SYN or TCP SYN/ACK packets from the same host that carry port numbers matching the stored encapsulated port numbers. Otherwise, process TCP SYN or TCP SYN/ACK packets as normal.

This method ensures that the TCP SYN / SYN/ACK handshake is visible to middleboxes and allows to immediately switch back to raw TCP communication in case of failures. If implemented on both sides as described above and no TiU SYN or TiU SYN/ACK packet arrives, yet a TCP SYN or TCP SYN/ACK packet does, this can only mean that the other host does not support TiU, a UDP packet was dropped, or the UDP and TCP packets were reordered in transit. Reordering in the host (e.g., a server responding to a TCP SYN before it responds to a TiU SYN) can be a problem for similar methods (e.g. [RFC6555]), but it can be eliminated by prescribing the processing order as above.

Because TCP does not preserve message boundaries and the size of the TCP header can vary depending on the options that are used, it is also no problem to precede the TCP header in the UDP packet with a different header (e.g. SPUD [I-D.hildebrand-spud-prototype]) without exceeding the known MTU limit. When creating a TCP segment, a TCP sender needs to consider the length of this header when calculating the segment size, just like it would consider the length of a TCP option. For this to work, the usage of other headers such as SPUD in-between the UDP header and the TiU header must therefore be known to both the sender-side and receiver-side code that processes TiU.

5. Coupled congestion control

For each TCP connection c, the algorithm described below receives cwnd and ssthresh as input and stores the following information:

- the Connection ID.
- a priority P(c) -- e.g., an integer value in the range from 1 (unimportant) to 10 (very important).
- The previously used cwnd used by the connection c, ccc_cwnd(c).
- The previously used ssthresh used by the connection c, ccc_ssthresh(c).

Three global variables S_CWND, S_SSTHRESH and S_P are used to represent the sum of all the ccc_cwnd values, ccc_sshtresh values and priorities of all TCP connections, respectively. S_CWND and S_SSTHRESH are used to update the cwnd and ssthresh values for all connections.

5.1. Example algorithm

This algorithm emulates the behavior of a single TCP connection by choosing one connection as the connection that dictates the increase / decrease behavior for the aggregate. It was designed to be as simple as possible. In the algorithm description below, abbreviations are used to refer to the phases of TCP congestion control as defined in [RFC5681]: SS refers to Slow Start, CA refers to Congestion Avoidance and FR refers to Fast Recovery.

For simplicity, this algorithm refrains from changing cwnd when a connection is in FR. SS should not happen as long as ACKs arrive. Hence, the algorithm ensures that the aggregate’s behavior is only dictated by SS when all connections are in the SS phase.

(1) When a connection c starts, it adds its priority P(c) to S_P. If it is the very first connection that uses the outer UDP port number pair, it also sets S_CWND to its own cwnd. After that, the connection’s globally known cwnd and ssthresh values (ccc_cwnd(c) and ccc_ssthresh(c)) are updated, and the

connection updates its own cwnd and ssthresh values to be equal to ccc_cwnd(c) and ccc_ssthresh(c).

```
S_P = S_P + P(c)
ccc_cwnd(c) = P(c) * S_CWND / S_P
ccc_ssthresh(c) = ssthresh
if (S_SSTHRESH > 0)
  ccc_ssthresh(c) = P(c) * S_SSTHRESH / S_P
end if
// Update c's own cwnd and ssthresh for immediate use:
send ccc_cwnd(c) and ccc_ssthresh(c) to the connection c
```

(2) When a connection c stops, its entry is removed. S_P is recalculated.

(3) Every time the congestion controller of a connection c calculates a new cwnd, the connection calls UPDATE, which carries out the tasks listed below to derive the new cwnd and ssthresh values for all the connections. Since we intend to emulate the behavior of one connection, we designate one of the connections as the "Coordinating Connection" (CoCo). Whenever the coordinating connection calls UPDATE, S_CWND and S_SSTHRESH are additionally updated to reflect the current sum of all stored ccc_cwnd and ccc_ssthresh values. Initially, there is only one connection and this connection automatically becomes the CoCo. It updates S_CWND to its own cwnd and sets S_SSTHRESH to 0.

(4) WHEN a non-CoCo connection c CALLS UPDATE......

```
if(all of the connections including CoCo are in CA but c is in FR)
   c becomes the new CoCo.
else
   if(c is in CA or SS)
      c's cwnd is assigned its previously stored ccc_cwnd value.
```

(5) WHEN c(CoCo) CALLS UPDATE......

```
if (c is in CA)
    if(cwnd >= ccc_cwnd(c)) // cwnd has increased
       S_CWND = S_CWND + cwnd - ccc_cwnd(c)
    else
       S_CWND = S_CWND * cwnd / ccc_cwnd(c)
    end if
    ccc_cwnd(c) = P(c) * S_CWND / S_P
    ccc_ssthresh(c) = ssthresh
    if (S_SSTHRESH > 0)
       ccc_ssthresh(c) = P(c) * S_SSTHRESH / S_P
    end if
    // Update c's own cwnd and ssthresh for immediate use:
    send ccc_cwnd(c) and ccc_ssthresh(c) to the connection c
end if

else if (c is in FR)
    S_SSTHRESH = S_CWND/2

else if (c is in SS)
    if (all other connections are in SS)
       S_SSTHRESH = S_CWND/2
       S_CWND = S_CWND * cwnd / ccc_cwnd(c)
       ccc_cwnd(c) = P(c) * S_CWND / S_P
       // Update c's own cwnd for immediate use:
       send ccc_cwnd(c) to the connection c
    else
       make any other connection which is not in SS the CoCo
    end if
end if
```

6. Usage considerations

TiU cannot work with applications that require the Urgent pointer (which is not recommended for use by new applications anyway [RFC6093], but should be consider if TiU is implemented in a way that allows it to be applied onto existing applications; telnet is a well-known example of an application that uses this functionality). It enables use of TCP with methods such as SPUD [I-D.hildebrand-spud-prototype]. It can also be used as a method to experimentally test new TCP functionality in the presence of middleboxes that would otherwise create problems (as some have been known to do [Honda11]). TCP option space is getting scarce, in particular on TCP SYN and TCP SYN/ACK packets. Rather than stretching the Data Offset field on TCP SYN / TCP SYN/ACK packets (which was considered for TiU design), it is recommended to use one of the other proposed mechanisms to stretch option space, e.g. "Inner Space" [I-D.briscoe-tcpm-inner-space].

Reasons to use TiU include the benefits of [Che13] and [Den08] that were discussed in Section 1. TiU has the disadvantage of disabling ECMP for the TCP connections that it encapsulates. This can reduce the capacity usage of these TCP connections. It has the advantage of being able to apply coupled congestion control, which can provide precise congestion window assignment based on a priority. Other benefits of TiU's coupled congestion control are:

o Reduced average loss and queuing delay (because the competition between the encapsulated TCP connections is avoided)

o Even in the absence of prioritization, better fairness between the TiU-encapsulated TCP connections

o No need for new TiU connections to slow start up to a reasonable cwnd value that ongoing TiU connections already have: a connection can immediately be assigned its share of the aggregate's total cwnd. This can significantly reduce the completion time of short connections.

All of these benefits only play out when there are more than one TCP connections. Some of the benefits in the list above are more significant when some transfers are short. Moreover, short transfers are less likely than long ones to saturate the capacity of a path, reducing the chance to benefit from ECMP (which TiU eliminates). This makes the usage of TiU especially attractive in situations where some transfers are short.

7. Implementation status

The University of Oslo is currently working on a FreeBSD kernel implementation of TCP-in-UDP.

8. IANA Considerations

This document specifies a new TCP option that uses the shared experimental options format [RFC6994]. No value has yet been assigned for ExID.

This document requires a well-known UDP port (referred to as port XXX in this document). Due to the highly experimental nature of TiU, this document is being shared with the community to solicit comments before requesting such a port number.

9. Security Considerations

We have not thought about security yet. This will surely be fun!

10. Acknowledgement

This work has received funding from Huawei Technologies Co., Ltd., and the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

11.2. Informative References

[Che13] Cheshire, S., Graessley, J., and R. McGuire, "Encapsulation of TCP and other Transport Protocols over UDP", Internet-draft draft-cheshire-tcp-over-udp-00, June 2013.

[Den08] Denis-Courmont, R., "UDP-Encapsulated Transport Protocols", Internet-draft draft-denis-udp-transport-00, July 2008.

[EFCM] Savoric, M., Karl, H., Schlager, M., Poschwatta, T., and A. Wolisz, "Analysis and performance evaluation of the EFCM common congestion controller for TCP connections", Computer Networks (2005) , 2005.

[ETCP] Eggert, L., Heidemann, J., and J. Joe, "Effects of ensemble-TCP", ACM SIGCOMM Computer Communication Review (2000) , 2000.

[fse] Islam, S., Welzl, M., Gjessing, S., and N. Khademi, "Coupled Congestion Control for RTP Media", ACM SIGCOMM Capacity Sharing Workshop (CSWS 2014) and ACM SIGCOMM CCR 44(4) 2014; extended version available as a technical report from http://safiquli.at.ifi.uio.no/paper/fse-tech-report.pdf , 2014.

[Honda11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and H. Tokuda, "Is it still possible to extend TCP?", Proc. of ACM Internet Measurement Conference (IMC) ’11, November 2011.

[I-D.briscoe-tcpm-inner-space] Briscoe, B., "Inner Space for TCP Options", draft-briscoe-tcpm-inner-space-01 (work in progress), October 2014.

[I-D.hildebrand-spud-prototype] Hildebrand, J. and B. Trammell, "Substrate Protocol for User Datagrams (SPUD) Prototype", draft-hildebrand-spud-prototype-03 (work in progress), March 2015.

[I-D.ietf-rmcat-coupled-cc] Islam, S., Welzl, M., and S. Gjessing, "Coupled congestion control for RTP media", draft-ietf-rmcat-coupled-cc-00 (work in progress), September 2015.

[RFC1078] Lottor, M., "TCP port service Multiplexer (TCPMUX)", RFC 1078, DOI 10.17487/RFC1078, November 1988, <http://www.rfc-editor.org/info/rfc1078>.

[RFC2140] Touch, J., "TCP Control Block Interdependence", RFC 2140, DOI 10.17487/RFC2140, April 1997, <http://www.rfc-editor.org/info/rfc2140>.

[RFC3124] Balakrishnan, H. and S. Seshan, "The Congestion Manager", RFC 3124, DOI 10.17487/RFC3124, June 2001, <http://www.rfc-editor.org/info/rfc3124>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <http://www.rfc-editor.org/info/rfc5681>.

[RFC6093] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <http://www.rfc-editor.org/info/rfc6093>.

[RFC6356] Raiciu, C., Handley, M., and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols", RFC 6356, DOI 10.17487/RFC6356, October 2011, <http://www.rfc-editor.org/info/rfc6356>.

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <http://www.rfc-editor.org/info/rfc6555>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <http://www.rfc-editor.org/info/rfc6994>.

Authors’ Addresses

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: michawe@ifi.uio.no

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no

Kristian Hiorth
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: kristahi@ifi.uio.no

Jianjie You
Huawei
101 Software Avenue, Yuhua District
Nanjing 210012
China

Email: youjianjie@huawei.com